Numerical Optimization using PETSc/TAO

Alp Dener, Todd Munson, Hong Zhang

PDE-Constrained Optimization

$$\begin{aligned} \min_{u,v} \quad & f(u,v) \\ \text{subject to} \quad & g(u,v) = 0 \\ & c(u,v) \geq 0 \end{aligned}$$

u: state variables
v: design variables

- **g**: state equations
  - discretization of partial differential equation given design
- **c**: constraints
  - includes constraints on both the state and design

FASTMATH

PDE-Constrained Optimization Applications

- Applications include
  - Inverse problems
  - Parameter estimation
  - Design optimization
- Several packages available
  - Toolkit for Advanced Optimization (PETSc/TAO)
  - Rapid Optimization Library (ROL)

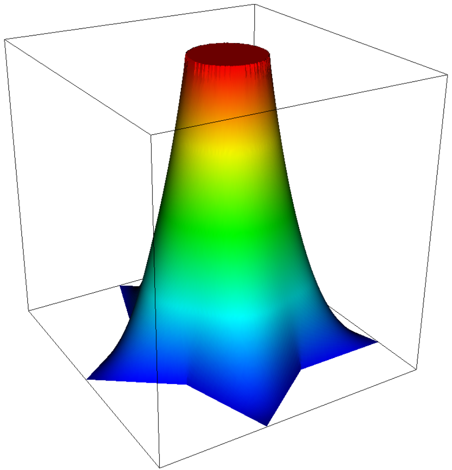
Outline

- Bound-constrained optimization methods
  - Best method to apply is problem dependent
    - TAO provides many choices for nonlinear problems
  - Generally, use second derivatives for best performance
  - MFEM can easily be used for optimization problems
- Dynamic optimization problems using adjoints

FASTMATH

The Obstacle Problem

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \int_\Omega |\nabla u|^2 dx \\ \text{subject to} \quad & u(x) \geq \phi(x) \; \forall \; x \in \Omega \\ & u(x) = 0 \; \forall \; x \in d\Omega \end{aligned}$$

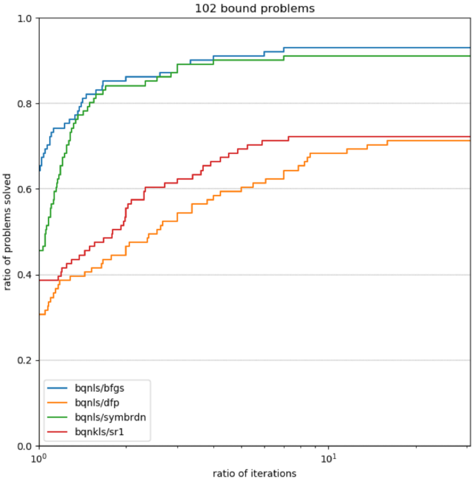


TAO Bound-Constrained Algorithms

$$\begin{array}{ll} \min\limits_{s} & \nabla f(u^k)^T s + \frac{1}{2} s^T H^k s \\ \text{subject to} & x^k + s \geq 0 \end{array}$$

- Approximate the objective function
  - Quasi-Newton (-tao_type bqnls) uses approximation
  - Newton-Krylov (-tao_type bnls) uses Hessian
- Approximate the set of active bounds
- Solving a linear system of equations for direction
- Ensure global convergence
  - Line search
  - Trust region

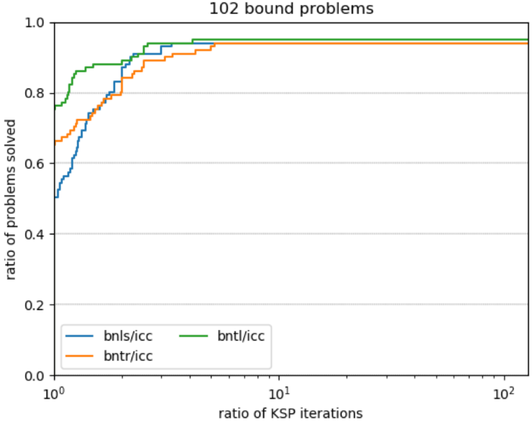
PETSc/TAO: Quasi-Newton Methods (bqnls)



- Multiple limited-memory QN approximations implemented as PETSc Mat objects
- Relative performances compared on full set of bound constrained CUTEst problems
- TAO QN algorithms can seamlessly change methods

PETSc/TAO: Newton-Krylov Methods (bnls)

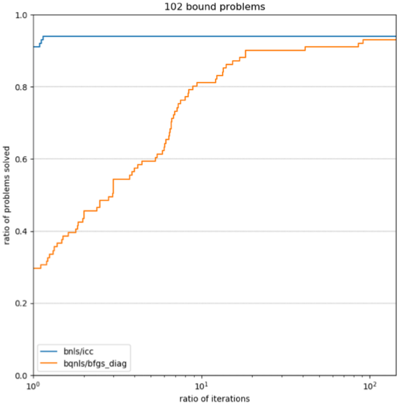
- Globalization strategy makes very little difference

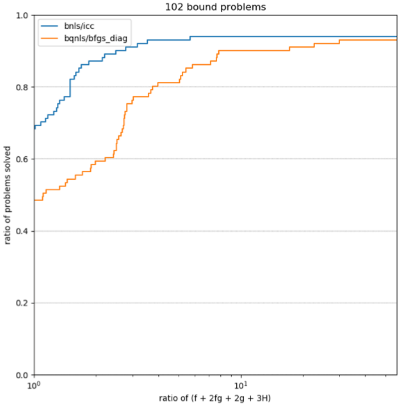


FASTMATH

Effect of Second-Order Information

- NK outperforms QN in both nonlinear iterations and function/gradient/Hessian evaluations

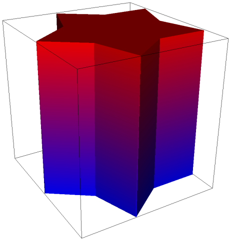
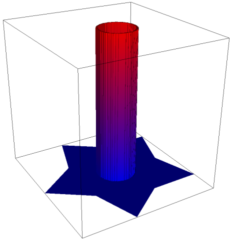
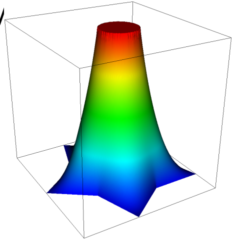




The Obstacle Problem: MFEM-TAO Integration

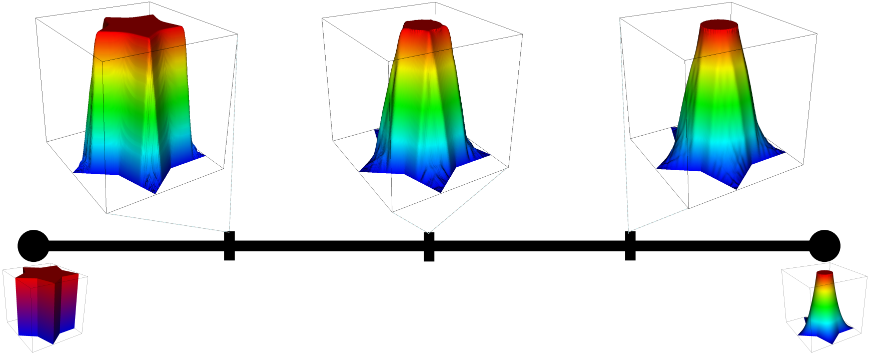
$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \int_\Omega |\nabla u|^2 dx \\ \text{subject to} \quad & u(x) \geq \phi(x) \; \forall \, x \in \Omega \\ & u(x) = 0 \; \forall \, x \in d\Omega \end{aligned}$$

- Newton-Krylov method
- Objective, gradient and Hessian evaluations through FEM

The Obstacle Problem: Quasi-Newton Solution

- QN solution converges in 292 iterations
- A lot of computational effort is spent making small changes near the obstacle



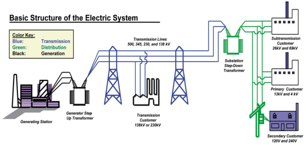
Transition to the Hands On Lesson

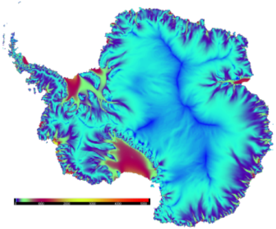
https://xsdk-project.github.io/ATPESC2018HandsOnLessons/lessons/obstacle_tao/

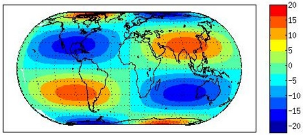
FASTMATH

Adjoints are key ingredients in PDE-constrained optimization

Research interests have been shifting beyond modelling and simulation of a physical system to **outer-loop applications** such as **PDE-constrained optimization**, optimal design and control, uncertainty quantification etc.







Solving optimization problems often requires to compute derivatives of a functional, which can be computed efficiently with **adjoints**.

What is PDE-constrained optimization?

Goal

Solve the discrete optimization problem

$$\underset{p,u}{\text{minimize}}\ \mathcal{J}(u, p)$$

$$\begin{aligned} \text{subject to} \quad & c(u, p, t) = 0 && \text{(PDE constraint)} \\ & g(u, p) = 0 && \text{(equality constraints)} \\ & h(u, p) \leq 0 && \text{(inequality constraints)} \end{aligned}$$

where

- $\mathcal{J}$ is the objective functional
- c represents the discretized PDE equation
- $u \in \mathcal{R}^n$ is the PDE solution state
- $p \in \mathcal{R}^m$ is the parameters

Because the dimension of u can be really high, a reduced formulation is often used.

$$\mathcal{J}(p) = \mathcal{J}(u(p), p)$$

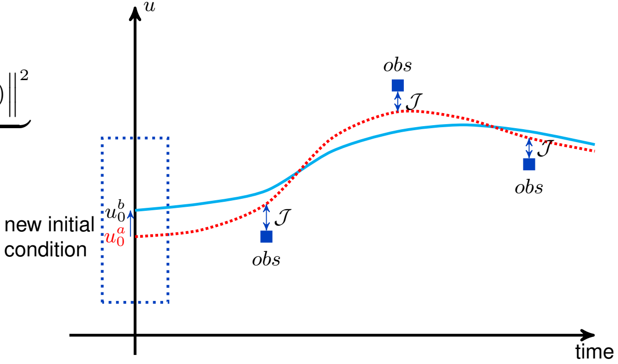
FASTMATH

An example: data assimilation

The objective function of data assimilation is

$$\mathcal{J}(u(u_0), u_0^a) = \underbrace{\frac{1}{2}\|Qu - d\|^2}_{\text{observation error}} + \underbrace{\frac{\alpha}{2}\left\|L(u_0^a - u_0^b)\right\|^2}_{\text{background error}}$$

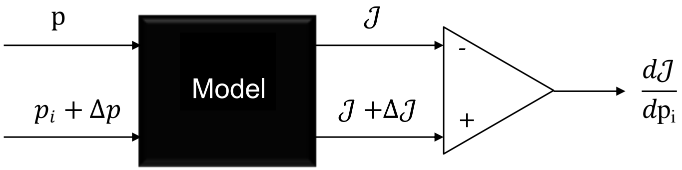
- ☐ state variable y, control or design variable u, data d
- ☐ Q is observation operator
- ☐ L is cost functional for design
- ☐ α is tradeoff between cost of design and fitting data



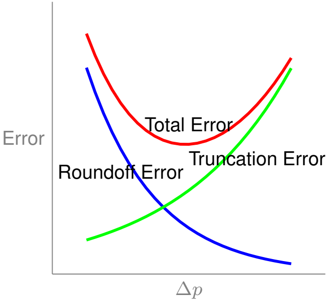
- Physical interpretation: Determine the optimal initial conditions for a numerical model that minimizes the difference between the forecast and the observations
- A regulization term is often added to the cost functional to ensure existence and uniqueness
- Gradient-based optimization algorithms require local derivatives (sensitivities)

FASTMATH

Computing sensitivities: finite differences

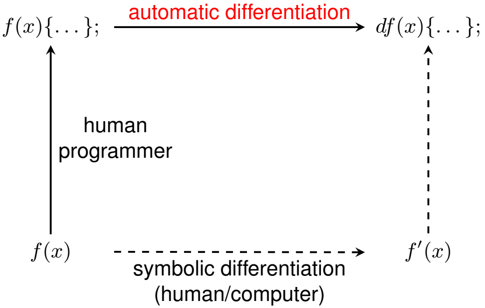


- Easy to implement
- Inefficient for many parameter case, due to one-at-a-time
- Possible to perturb multiple parameters simultaneously by using graph coloring
- Error depends on the perturbation value Δp



Computing sensitivities: automatic differentiation

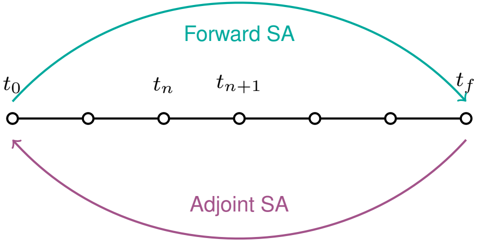
- AD can evaluate the sensitivities for an arbitrary sequence of computer codes
- Difficulties of low-level AD
  - pointers
  - dynamic memory
  - directives
  - function calls from external libraries
  - iterative processes (e.g. Newton iteration)
  - non-smooth problems



Forward and adjoint sensitivity analysis (SA) approaches

We compute the gradients by differentiating the time stepping algorithm, e.g. backward Euler ($y_{n+1} = y_n + h\,\mathbf{f}(t_{n+1}, y_{n+1})$)

$$\mathbf{S}_{\ell,n+1} = \mathbf{S}_{\ell,n} + h\,\mathbf{f_y}(t_{n+1}, y_{n+1})\mathbf{S}_{\ell,n+1}$$

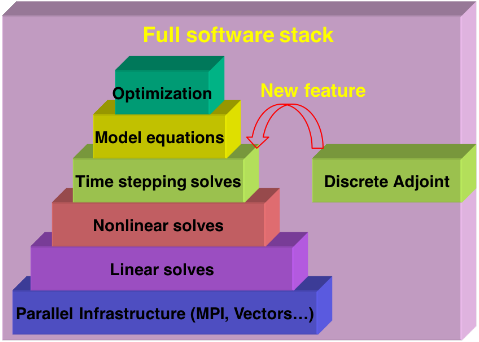


$$\lambda_n = \lambda_{n+1} + h\,\mathbf{f_y}\,(t_{n+1}, y_{n+1})^T\,\lambda_n$$

	Forward	Adjoint
Best to use when	# of parameters $<<$ # functionals	# of parameters $>>$ # of functionals
Complexity	$\mathcal{O}$ (# of parameters)	$\mathcal{O}$ (# of functionals)
Checkpointing	No	Yes
Implementation	Medium	High
Accuracy	High	High

Adjoint integration with PETSc

- PETSc: open-source numerical library for large-scale parallel computation
  https://www.mcs.anl.gov/petsc/
- $\sim$ 200,000 yearly downloads
- Portability
  - 32/64 bit, real/complex
  - single/double/quad precision
  - tightly/loosely coupled architectures
  - Unix, Linux, MacOS, Windows
  - C, C++, Fortran, Python, MATLAB
  - GPGPUs and support for threads
- Extensibility
  - ParMetis, SuperLU, SuperLU_Dist, MUMPS, HYPRE, UMFPACK, Sundials, Elemental, Scalapack, UMFPack...
- Toolkit
  - sequential and parallel vectors
  - sequential and parallel matrices (AIJ, BAIJ...)
  - **iterative solvers and preconditioners**
  - **parallel nonlinear solvers**
  - **adaptive time stepping (ODE and DAE) solvers**



Other software for adjoints and related functionality

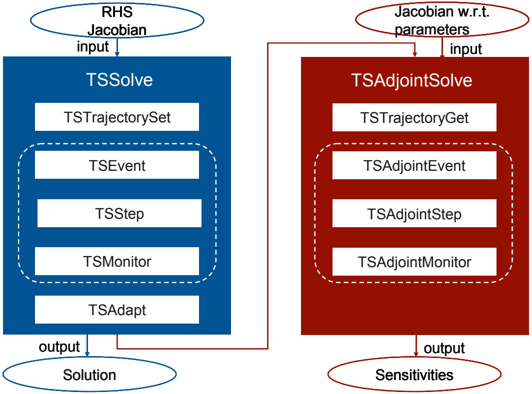
Also available in:

- SUNDIALS
- Trilinos

This presentation focuses on experiences in PETSc.

TSAdjoint Interfaces are smilar to TS interfaces

- Designed to reuse functionalities (implemented in PETSc or provided by users)
- Aim for general-purpose solutions
- Support both explicit and implicit methods and timestep adaptivity
- Allow multiple cost functionals



Optimal checkpointing for given storage allocation

- Minimize the number of recomputations and the number of reads/writes by using the **revolve** library of Griewank and Walther
  - **Revolve** is designed as a top-level controller for time stepping
  - TSTrajectory consults **revolve** about when to store/restore/recompute
- Incorporate a variety of single-level and two-level schemes for offline and online checkpointing
  - existing algorithms work great for RAM only checkpointing
  - optimal extension for RAM+disk (work in progress)

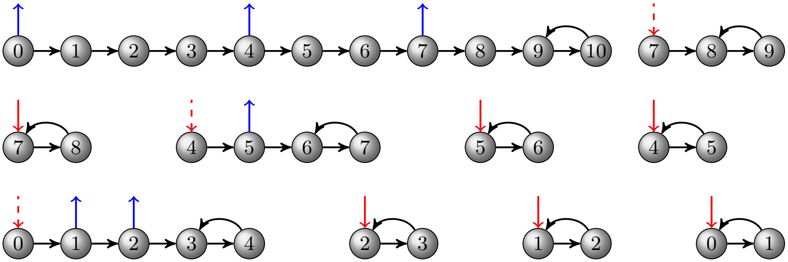
An optimal schedule given 3 allowable checkpoints in RAM:

blue arrow: store a checkpoint

red arrow: restore a checkpoint

black arrow: a step

circle: solution



Validating Jacobian and sensitivity is critical for optimization

- PETSc and TAO (optimization component in PETSc) can test hand-coded Jacobian and gradients against finite difference approximations on the fly
- Jacobian test: `-snes_test_jacobian`

```
Norm of matrix ratio 2.83894e-08, difference 1.08067e-05 (user-defined state)
Norm of matrix ratio 3.36163e-08, difference 1.31068e-05 (constant state -1.0)
Norm of matrix ratio 3.33553e-08, difference 1.3005e-05 (constant state 1.0)
```

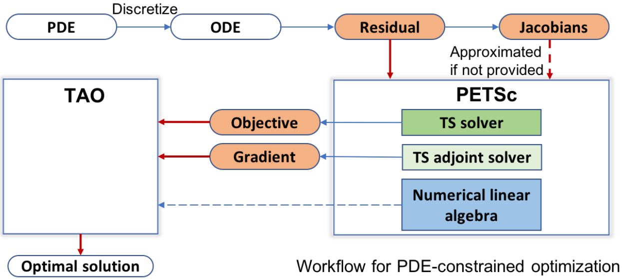
- Gradient test: `-tao_test_gradient`

```
||fd|| 0.168434, ||hc|| = 1.18456, angle cosine = (fd'hc)/||fd||||hc|| = 0.987391
2-norm ||fd-hc||/max(||hc||,||fd||) = 0.859896, difference ||fd-hc|| = 1.01859
max-norm ||fd-hc||/max(||hc||,||fd||) = 0.853218, difference ||fd-hc|| = 0.311475
```

- `-snes_test_jacobian_view` and `-tao_test_gradient_view` can show the differences element-wisely
- Nonlinear solve is not very sensitive to the accuracy of Jacobian, but adjoint solve needs accurate Jacobian

FASTMATH

Solving dynamic constrained optimization



Workflow for PDE-constrained optimization

Set up TAO:

- Initial values for the variable vector
- Variable bounds for bounded optimization
- Objective function
- Gradient function
- Hessian matrix for Newton methods (optional)

Set up ODE solver and adjoint solver:

- ODE right-hand-side function and Jacobian
- Additional Jacobian w.r.t parameters if gradients to the parameters are desired.
- ODE Initial condition
- Terminal conditions (initial values for adjoint variables) for the adjoint variables

FASTMATH

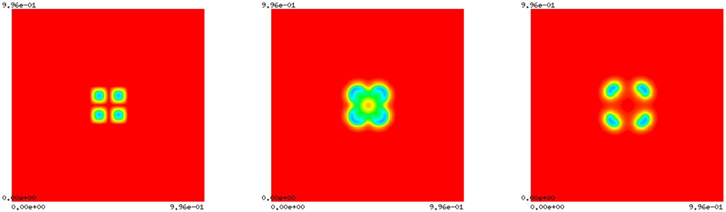
Hands-on: an inverse initial value problem

$$\underset{U_0}{\text{minimize}} \; \|U(t_f) - U^{ob}(t_f)\|_2$$

$$\text{subject to} \quad \frac{d\mathbf{u}}{dt} = D_1 \nabla^2 \mathbf{u} - \mathbf{u}\mathbf{v}^2 + \gamma(1 - \mathbf{u})$$

$$\frac{d\mathbf{v}}{dt} = D_2 \nabla^2 \mathbf{v} + \mathbf{u}\mathbf{v}^2 - (\gamma + \kappa)\mathbf{v}$$

where $U = [\mathbf{u}; \mathbf{v}]$ is the PDE solution vector, U_0 is the initial condition. The reaction and diffusion of two interacting species can produce spatial patterns over time.



(a) t=0 sec (b) t=100 sec (c) t=200 sec

Interpretation Given the pattern at the final time, can we find the initial pattern?

Link to Hands-on Lesson

https://xsdk-project.github.io/ATPESC2018HandsOnLessons/lessons/adjoint

Tips and advice

- Jacobian can be efficiently approximated using finite difference with coloring (`-snes_fd_coloring`); particularly convenient via `DMDA`
- Most of the difficulties stem from mistakes in the hand-coded Jacobian function; make sure to validate it carefully
- Use direct solvers such as SuperLU and MUMPS for best accuracy (but not scalability) of the gradients
- Use `-tao_monitor` `-ts_monitor` `-ts_adjoint_monitor` `-snes_monitor` `-log_view` for monitoring the solver behavior and profiling the performance
- `-malloc_hbw` allows us to do the computation using MCDRAM and checkpointing using DRAM on Intel's Knights Landing processors (Argonne's Theta, NERSC's Cori)
- Check the user manual and the website for more information, and ask questions on the mailing lists

Takeaways

- PETSc and TAO help you rapidly develop parallel code for dynamic constrained optimization
- Adjoint as an enabling technology for optimization
- PETSc offers discrete adjoint solvers that take advantage of highly developed PETSc infrastructure: MPI, parallel vectors, domain decomposition, linear/nonlinear solvers
- Requires minimal user input, and reuses information provided for the forward simulation
- Advanced checkpointing, transparent to the user
- Validation for Jacobian and gradients using finite differences

Thank you!